学号 2019300003008

学号 2019302180116

武汉大学本科论文

DEfensor：
基于 eBPF 的去中心化信息流控制系统

院（系）名　称：国家网络安全学院

专　业　名　称：信息安全

学　生　姓　名：邓浚泉 仇文彬

指　导　教　师：严飞　副教授

二○二二年六月

郑 重 声 明

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名：_____ 日期：_____

摘　　要

去中心化信息流控制（DIFC）是一种访问控制方法，它允许程序编写者控制程序与其它信息实体之间的数据流。当应用于隐私性时，DIFC 允许不受信任的程序处理私密数据，而受信任的程序控制该数据的传出。当应用于完整性时，DIFC 允许受信任程序保护不受信任的程序免受意外的恶意输入。在上面两种情况下，只有受信任程序中的安全漏洞才可能导致安全问题，从而在普遍情况下做到了对信息流的安全控制。

我们提出了一个新的去中心化信息流控制系统 DEfensor，它以进程为粒度控制信息流，以标准读写接口为粒度执行安全策略，简化了 DIFC 在现有应用中的使用，并解决了过往 DIFC 系统可移植性差、性能低下、引入内核风险的缺点。DEfensor 在内核空间中使用扩展的伯克利包过滤器（eBPF）观测进程行为并执行安全策略，在用户空间中运行引用监视器管理信息流状态和安全规则。这一系统被认为是行之有效的。

关键词：信息流控制系统；IFC；DIFC；eBPF；系统安全模型

目　　录

1 介绍

随着现代应用程序在规模、复杂性和对第三方软件的依赖性方面的增长，它们更容易受到安全缺陷的影响。分散信息流控制（DIFC）[1] 是经典信息流控制 [2–4] 的一个变种，它可以提高复杂应用程序的安全性，甚至是程序存在潜在的漏洞的前提下。现有的 DIFC 系统主要作为编程语言抽象 [5] 运行，或者集成到操作系统的通信原语中 [6, 7]。这些方法在信息流的细粒度控制和高性能方面具有一定优势，但需要改变应用程序的开发方式，不利于应用程序的维护。作为替代，我们的 DEfensor 系统提供了进程级的 DIFC 系统，作为对现有操作系统中通信原语的最小扩展，使 DIFC 系统能够让程序员在使用熟悉的语言、工具和操作系统抽象的前提下进行搭建开发。

DEfensor 系统以进程的粒度提供 DIFC，并通过运行在用户态的引用监视器、在内核态的 eBPF 钩子进行协同运作，将 DIFC 控制系统在标准通信抽象（如管道、套接字和文件描述符）的粒度上集成实现，同时提供便利的用户接口，使程序员可以在较少地改动代码的前提下进行 DIFC 系统的搭建开发。

DEfensor 系统将系统进程分为两种，其一是不受信任的进程，执行操作系统的大部分计算任务，并在和受信任进程通信时受到 DEfensor 系统的限制。在平时情况下，DEfensor 对不受信任进程是透明的。另一种是受信任的进程，在应用程序运行时强制执行 DIFC 策略，并了解 DIFC 系统的存在，设置隐私和完整性控制来约束不受信任的进程。同时，受信任的进程还有权选择性地违反经典的信息流控制，例如将私有数据降密（以方便从系统中导出），或将数据加密为高完整性标签数据。此权限根据应用程序策略分布在受信任进程之间，使其分散而达到“分散式”信息流控制。尽管受信任代码中的漏洞可能会导致危害，但应用程序中其他地方的安全漏洞却不会导致危害，同时即使在应用程序扩展的过程中，受信任代码也可以保持相对独立和简洁，以防止出现漏洞。

DIFC 模型面临的一个核心问题是如何在不影响进程原有功能的前提下实现信息流控制。传统的进程接口充斥着容易泄漏信息的通道，如网络套接字等不可信的信道。系统可以简单地禁止这些通道，将进程通信接口限制为具有明显和可控信息流的系统调用，但这种方法会使许多库函数无法使用，迫使非受信任进程修改代码以适应 DIFC 系统。DEfensor 的解决方案是将信道的拦截过程分为建立

和通信两步，以进程建立通信的每个信道资源作为最小粒度，包括管道、套接字、文件和网络连接等，并在知晓信道两端实体的前提下进行权限判断，从而可以在允许进程建立通信的同时，拦截部分信道的通信，并保证大部分非受限信道的成功通信，使进程在不需修改自身代码的前提下依然可以在 DIFC 系统中运行。

为了性能和可移植性，我们利用 eBPF 和处于用户空间的引用监视器构建了 DEfensor，并在 Linux 上运行。与以前将 DIFC 作为新内核设计的一部分提供的系统（如 Asbesto[6] 和 Histar[7]）不同，DEfensor 利用现有的大量工作（如 eBPF、Tracepoint、kprobe、LSM、KRSI 等工具）来进行实现，这些现有工具在最新的 Linux 内核（高于 5.13）都已得到支持。此外，作为核心技术的 eBPF 将会在未来在 Windows 系统上提供支持，运用 eBPF 技术可以在未来的工作中提供跨平台性。

2 DEfensor 中的信息流

本节介绍 DEfensor 的去中心化信息流控制方法，DEfensor 的 DIFC 模型与传统的 IFC 类似[4]。

2.1 标签体系

DEfensor 使用标签跟踪系统中数据的流通。假设 F 是一个很大的 token 集合，这个集合叫做全标签集合，任意标签 t 都是它的元素。

标签没有固有的含义，但进程通常将每个标记与私密性或完整性相关联。例如，Tag b 可以用来标记 Bob 的私密数据。任意一个进程 p 都有两个标签集，私密标签集 S_p 和完整标签集 I_p。如果标签 $t \in S_p$，系统就会做这样的保守假设：p 看到了某些带有 t 标签的私密数据。对完整性而言，如果 $t \in T_p$，则 p 的任意输入都被认为可以对标记数据的完整性负责。文件（和其他对象）也有私密和完整标签。

尽管任何标签都可以出现在任何标签集中，但实际使用中标签只在私密标签集中使用或只在完整标签集中使用。因此，标签可以直接地分类为“私密标签”和“完整标签”。

接下来将用几个示例来说明 DEfensor 的信息流。

示例：保护私密性 Bob 希望在个人电脑上保存一些私密文件，但他可能下载到一个恶意的文本编辑器。这个文本编辑器会将私密文件发布到公共网站，或者上传到黑客的服务器。在典型的操作系统安全语境中，Bob 只能寄希望于文本编辑器不会泄露他的数据，选择从受信任的渠道下载软件。

通过信息流控制，Bob 可以限制编辑器的行为，而无需考虑它是否是恶意的。假设标签 b 用来代表 Bob 的私密数据，Bob 可以对所有不可信的进程 p 运用以下规则：

1. 如果 p 读了他的私密文件，那么 $b \in S_p$.
2. 如果 $b \in S_p$，那么 p 只能向符合 $b \in S_q$ 的进程 q 或文件 q 传输信息。
3. p 不能将标签 b 从自己的私密集合 S_p 中移除。
4. 如果 $b \in S_p$，p 不能向不受控制的信道传输信息。（比如因特网）

如果所有这些条件都满足，编辑器就不能从系统中泄露 Bob 的私密数据。

示例：保护完整性 假设 Bob 在他的机器上拥有管理员权限，允许他编辑敏感文件。然而，其他用户不断更新库并下载新软件。这可能引入恶意编辑器或恶意的依赖库。因此 Bob 使用编辑器编辑敏感文件时，担心编辑器不会忠实地按照他的指令操作，而是偷偷地篡改文件内容。

他同样可以运用信息流控制来消除这种担心。假如完整标签 v 代表需要保护的文件数据，而有些进程能够将 v 赋予文件或其它进程。Bob 对其它进程 p 运用以下规则：

1. 如果 p 编辑敏感文件，那么 $v \in I_p$.
2. 如果 $v \in I_p$，那么 p 不能从不符合 $v \in I_q$ 的进程 q 或文件 q 中读取信息。只有未被污染的文件（在这个例子中就是 Bob 信任的可执行文件和库）有 v 标签。
3. p 不能主动将 v 加到 I_p 中。
4. 如果 $v \in I_p$，p 不能从不受控制的信道接收信息。

如果所有条件满足，Bob 知道所有编辑敏感文件的编辑器都是未被污染的。

2.2 去中心化的权限

在传统的信息流控制中，只有受信任的安全管理员才能创建新标签，从私密标签集中移除标签（*降密*，*declassify*），或将标签添加到完整标签集（*背书*，*endorse*）。在 DEfensor DIFC 中，任何进程都可以创建新的标签。而创建标签的进程默认有权降密和或背书这个标签代表的信息。

DEfensor 使用两种能力表示权限。对于任意标签 t，它拥有两种能力，t^+ 和 t^-. 每个进程 p 都有一个能力集合 O_p。如果 $t^+ \in O_p$，则 p 拥有 t^+ 能力，它就有权将 t 添加到它的标签集中；如果 $t^- \in O_p$，则 p 拥有 $t-$ 能力，它有权将 t 从它的标签集中移除。就私密性而言，t^+ 能力允许进程添加私密标签，授予自己接收私密数据的权限。而 t^- 让它可以移除私密标签，降密它所看到的任何私密数据。就完整性而言，t^- 让进程可以移除完整标签，允许它从低信任的信息来源接收数据。而 t^+ 让它可以添加完整标签，使进程为 t 代表的完整性背书。一个同时拥有 t^+ 和 t^- 的进程拥有双重权限，可以完全控制标签 t 是否出现在它的标签集中。集合 $D_p = \{t | t^+ \in O_p \; and \; t^- \in O_p\}$ 表示 p 具有双重权限的所有标签。

任何进程 p 都能创建标签。假设创建的标签是 t，系统就会生成一个随机的

token 用于表示标签 t，并设置 $O_p \leftarrow O_p \cup \{t^+, t^-\}$，为 p 授予 t 的双重权限。

DEfensor 还支持一个全局能力集合 O. 每个进程都拥有 O 中的所有能力，系统会强制令 $O \subseteq O_p$. 只有标签的创建者能决定是否将标签的 t^+ 能力或 t^- 能力加入全局能力集。进程能问询一个能力是否在全局能力集 O 中，但它无法枚举 O，以防止信息泄露。

两个进程只要能互相通信，就能传递能力。同时，进程也能自由地放弃不在全局能力集 O 中的能力。

对于一个标签集 T，我们定义 $\{T\}^+$ 为 $\{t^+ | t \in T\}$，$\{T\}^-$ 同理。

示例：保护私密性 Bob 可以使用一种称为 "*传出保护*" 的策略来维护其私人数据的私密性。Bob 的一个进程创建了私密标签 b，用于标记他的私人数据；在创建标签时，b^+ 会被加到全局能力集 O 中，但只有受信任的进程会得到 b^-. 因此，任何进程 p 都可以把 b 添加 S_p 中，从而读取 Bob 的私密数据。但只有拥有 b^- 能力的进程（即 Bob 信任的进程）可以降密数据并将其传出系统。

一个更严格的策略称为 "*读保护*"。进程创建一个私密标签 t，但不将 t^+ 和 t^- 添加到全局能力集 O 中。通过控制 t^+，创建进程可以限制哪些进程可以查看 t 代表的私密数据，以及限制哪些进程可以降密这些数据。读保护可以防止机密数据通过隐蔽通道泄漏，比如基于时间的隐蔽信道。[7]

示例：保护完整性 另一种策略叫做 "*完整保护*"。一个进程创建了完整标签 v，在创建时，v^- 被添加到全局能力集 O 中。现在，任何进程 p 都可以将 v 从 I_p 中移除。但只有创建进程具有 v^+，可以把 v 添加到完整标签集中，从而为信息背书。这个创建标签的进程就类似于一个认证者。Bob 需要用他的编辑器编辑受保护的文件，这个认证进程就执行 fork，创建一个具有 v 标签的新进程。孩子进程放弃了 v^+ 能力，调用 exec 运行编辑器。于是，编辑器 p 就有 $v \in I_p$ 且 $v^+ \notin O_p$，它只能读取高完整性级别的文件（无论是二进制文件、库文件还是配置文件），因此不会受污染文件的影响。

这三种策略：传出保护、读保护和完整保护，代表了标签的常规用法。当然，其它策略也是可能的。

2.3 安全性

DEfensor 模型假设许多进程在同一台机器上运行，并通过信道进行通信。该模型的目标是通过控制通信信道和标签变更来控制信息流安全。我们有以下定义：

定义 2.3.1：在 DEfensor 模型中，一个系统是安全的，当且仅当所有允许的标签变更是安全的，并且所有允许的信息传输是安全的。

接下来具体考察标签变更与信息传输的安全。

标签变更安全 在 DEfensor 模型中（和 Flume 以及 HiStar 一样），只有进程 p 自身可以更改自己的 S_p 和 I_p。而且必须显式地请求这样的更改。其他模型允许进程的标签通过接收其它进程的消息而更改[6, 8, 9]，但隐式的标签更改可能会导致隐蔽通道[4, 7]。当进程请求更改标签集时，只有进程能力允许的标签更改是安全的：

定义 2.3.2：对于一个进程 p，它的能力集是 O_p，它的私密标签集或完整标签集是 L，而变更后的标签集是 L'。我们可以说标签变更是安全的，如果：

$$\{L' - L\}^+ \cup \{L - L'\}^- \subseteq O_p. \tag{2.1}$$

例如，假设进程 p 希望从 S_p 中移除标签 t，从而得到 $S'_p = S_p - \{t\}$. 这一移除只有在 p 拥有 t^- 能力时才安全。对于增添标签，同样的逻辑也成立，从而得出上述公式。

信息传输安全 信息流控制限制进程通信以防止数据泄漏。DEfensor 模型与经典 IFC 遵循着类似的规则，即 p 只能向满足以下条件的 q 发送消息：$S_p \subseteq S_q$（“no read up, no write down”[2]）并且 $I_q \subseteq I_p$（“no read down, no write up”[3]）。公式的含义是，q 含有 p 的所有私密性标签，p 才能传给 q；p 含有 q 的所有完整性标签，q 才能接收 p.

进程 p、q 有能力变更标签，那么变更后也可能重新获得通信许可。考虑这种情况：进程含有增加和删除某个标签的能力，它就可以临时删除/增加该标签，通信结束后再恢复原样。这种临时标签变更只是形式上的，可以认为这种情况能够直接通信。

定义 2.3.3：假如一个进程 p 想向 q 传输信息，D_p、D_q 是分别是 p、q 拥有双重能力的所有标签组成的集合，S_p、S_q 是 p、q 当前的私密标签集，I_p、I_q 是 p、q

当前的完整标签集。那么，p 能够向 q 传输，如果：

$$S_p - D_p \subseteq S_q \cup D_q \; and \; I_p \cup D_p \supseteq I_q - D_q. \tag{2.2}$$

这是因为，为了促成通信，在考虑私密性时，p 应当尽量减少所含私密标签，q 应当尽量增加所含私密标签。完整性同理。如果 D_p、D_q 为空，该公式就退化成了前面提到的基本公式。

这个公式隐含了这样一个过程：如果 p 一定要给 q 发送信息，它可以降密某些私密数据，或者背书某些完整数据。

2.4 信息实体

在 DEfensor 模型中，信息实体指所有含有信息的实体，进程、文件、信道等都是信息实体。针对不同的信息实体，DEfensor 会用不同的方式处理。

进程 进程是最基本的信息实体，它是 DEfensor 控制的主要对象。进程拥有信息，同时还拥有信息控制能力：它能主动地建立通信，还能根据能力集创建、修改或移除标签。更确切地说，每个进程 p 拥有三个集合，私密标签集 S_p，完整标签集 I_p 以及能力集 O_p。进程可以根据 2.2 节中描述的能力操作自身的标签集，同时可以与其它进程分享能力。

外部资源 任何不受 DEfensor 控制的数据接收者或数据源，如远程主机、用户终端、打印机等，都可以等效认为是一个具有空的私密标签集和完整标签集的无权进程 x：$S_x = I_x = \varnothing$ 且 $O_x = O$。因此，只有当进程 p 可以将其私密标签集减少到空集（唯一满足 $S_p \subseteq S_x$ 的集合）时，它才能向网络或控制台写入数据。同样，只有进程可以将其完整标签集减少到空集（唯一满足 $I_x \subseteq I_p$ 的集合），它才能从网络或键盘读取数据。

文件和目录 文件和目录可以等效认为是带有不可变标签集的进程，它们的私密标签集和完整标签集在创建时就固定了。一个进程 p 对一个文件 o 的写入就变成了从 p 向 o 的信息流；读取是从 o 到 p 的信息流。当一个进程 p 创建一个文件 o 时，p 可以指定 o 的标签集，只要 p 能够写文件 o。在许多情况下，p 还必须能写某些引用的文件或目录（例如，进程在创建文件时需要能写文件所在的目录）。

信道　信道可以等效认为是标签集由系统自动生成的进程。操作系统提供了许多 IPC 方式，譬如管道、FIFO、套接字、共享内存等，它们都可以被抽象为进程创建的文件——事实上在 Linux 内核中它们的确以文件的方式被实现。这样，进程通信的过程就变为了这样的过程：某个进程创建一个文件，其它进程打开这个文件，它们通过对这个文件读写来传输信息。对于文件，我们已经有了现成的处理方式。但信道与物理文件不同的地方在于，物理文件的标签集由进程在创建时赋予，但信道的标签集应当在进程建立通信时由系统自动协商。

出于自动协商的目的，系统会为每个信道 c 确定一个属主 w，信道的标签集和能力集就是属主的标签集和能力集：$S_c = S_w$，$I_c = I_w$ 且 $O_c = O_w$。而任意进程 p 从信道中读或写数据，就等同于 p 与属主进程 w 收或发数据，和一般的进程通信采用相同的安全判定策略。这一机制是为了解决多进程共用一个信道的问题，这种情况下的信息传输方向是难以控制的。属主的产生规则如下：当一个进程 a 发起信道时，信道的属主默认就是信道发起者 a。此后，不同的进程会进入或退出信道。如果 a 退出了信道，就顺延至下一位加入信道的进程 b 为信道属主。同样的，只要属主退出信道，下一进程就会顺延顶替。如果信道中最终没有进程了，也就可以认为这条信道不存在了。

3 DEfensor 的实现

过往的 IFC 系统要么在操作系统内核中实现，要么在用户空间实现[10–12]。内核实现意味着要大量修改操作系统内核代码，有可移植性差、实现难度大、引入内核风险的缺点。用户实现解决了这些问题，但也带来了性能低下、控制粒度粗糙的负面效果。为了解决现有 IFC 系统的问题，DEfensor 使用了一种兼顾内核和用户空间的实现，融合了两者的优点：既拥有极好的性能，又有高可移植性和精细的控制粒度。同时对内核的影响有限，风险可控。实现也相对简单。这都得益于 Linux 内核中引入的一项新技术——eBPF。

eBPF 是 Linux 内核 3.15 中引入的一项革命性技术，它能在内核中运行沙箱程序，而无需修改内核源码或者加载内核模块。将 Linux 内核变成可编程之后，就能基于现有的抽象层来打造更加丰富的基础设施软件，而不会增加系统的复杂度，也不会牺牲执行效率和安全性。eBPF 的核心构成是一个内核中运行的精简指令虚拟机、一个即时编译器（JIT）和一个加载验证器。eBPF 程序编写后，由加载验证器加载入内核。验证器会审查代码逻辑以确保其不会向内核引入风险（比如不能含有潜在的无限循环、只能访问规定的内存结构等）。然后，即时编译器实时地将 eBPF 程序翻译成机器码，在虚拟机中运行。这种模型的好处是，开发者能以最小的修改内核的代价实现只有内核空间才有可能实现的种种功能。换而言之，eBPF 在编程限制与内核能力间找到了一个平衡，这正是我们要寻找的绝佳工具。

3.1 系统框架

DEfensor 的设计具备经典的访问控制系统的特征，并吸收借鉴了其他 DIFC 系统实现（如 Flume[12]）的架构。图 3.1 展示了 DEfensor 实现的主要框架。

DEfensor 系统由一个运行在用户空间的引用监视器（Reference Monitor，RM）和运行在内核空间的 eBPF 程序组成。所谓引用监视器，指的是监视进程打开的文件和建立的信道（引用）并产生访问控制策略的控制器。在 DEfensor 系统中，引用监视器处于核心位置，所有信息实体的标签都由它管理，并受它控制。

引用监视器包含一个标签管理模块，一个引用管理模块，以及与用户进程和系统内核交互的接口。其中标签管理模块用于管理信息实体的标签，包括进程、文件及信道的私密标签集、完整标签集和能力集。引用管理模块用于管理进程加入

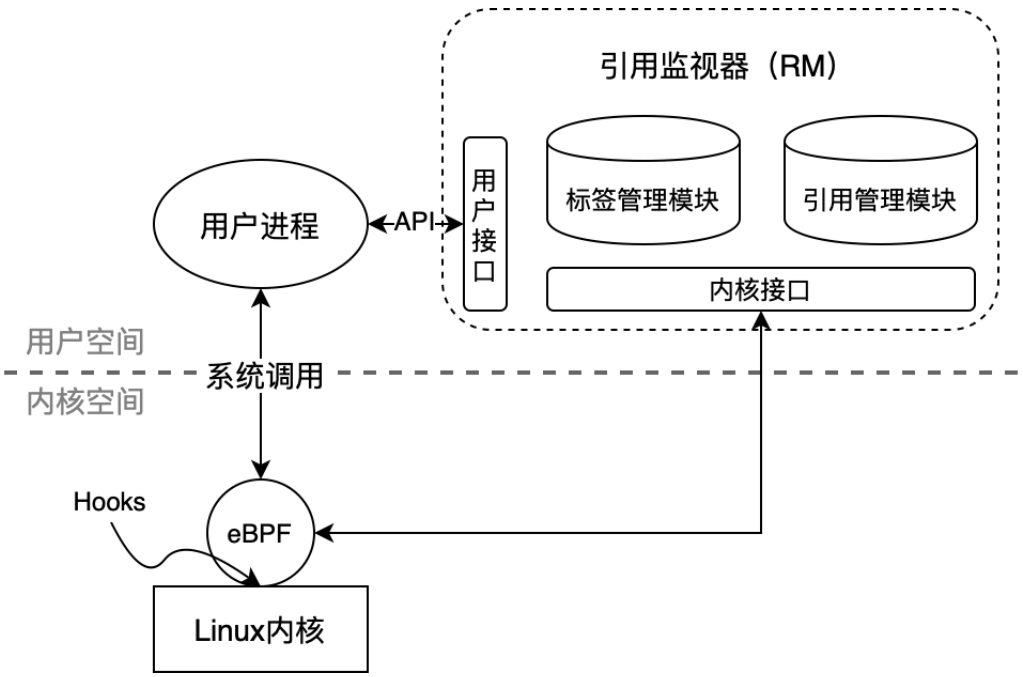

图 3.1　DEfensor 系统架构图

的信道（包括进程间通信和文件读写），以及信道的属性。譬如信道的属主和该信道成立所依赖的能力。标签管理模块和引用管理模块所做的工作主要是保持系统状态，它们管理维护着当前系统中的所有必要的信息实体信息，并向其它部分提供修改的接口。

为了与内核中的 eBPF 程序交互，引用监视器还包含了内核接口。它接受和分析 eBPF 程序提交的进程行为，并结合标签管理模块的标签信息，判断当前系统的信息流情况。信息流情况会被实时地交给引用管理模块，由它持久化存储。除了信息流情况，内核接口还负责根据 2.3 节中的安全条件，产生访问控制策略，然后交由 eBPF 程序执行。换而言之，它会判断系统各进程间能否建立通信，以及进程能否读写某个文件。用户接口则是一个面向用户进程的服务器程序。意识到 DEfensor 系统存在的程序（亦即那些主动使用 DEfensor，在程序逻辑中包含 DEfensor 调用的程序），可以通过调用用户接口提供的 API 函数，按需求创建、更改、分享标签。

3.2　状态存储

下面来详细考察 DEfensor 系统中的状态存储。图 3.2 是 DEfensor 中的状态存储模型。

为了保证信息流控制系统的响应速度，DEfensor 使用了内存数据库 Redis 作为存储解决方案。Redis 是一个以键-值对组织的数据库，DEfensor 建立了 11 个这样

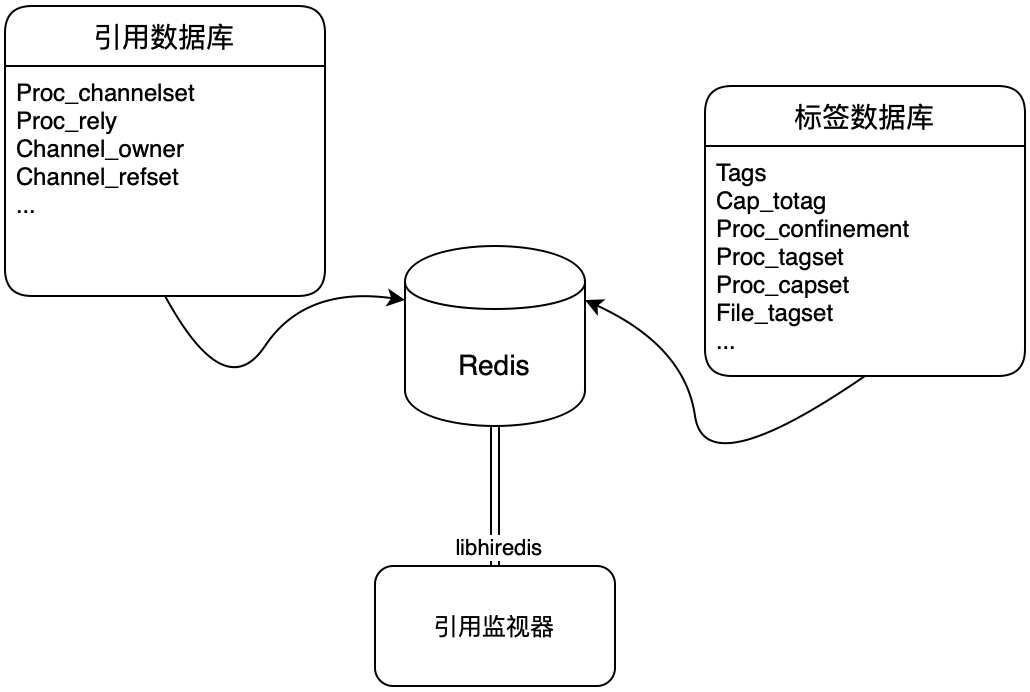

图 3.2　DEfensor 中的状态存储

的数据库，分别存储着信息实体的标签信息和引用信息。这些数据库根据被标签管理模块使用还是被引用管理模块使用分为两类。以图中的 Channel_refset 为例，它存储着引用这个信道的所有进程。在 Redis 中，它是一系列键为 channel_token，值为哈希表结构的表项。哈希表结构中的条目形如 (pid, fd) →permission，意为由进程 pid 和文件描述符 fd 唯一确定的信道引用，以及该信道引用是否被 DEfensor 系统允许读/写。

以上这些存储中大量使用 token 作为标识符。事实上，每个标签、能力、信道都有一个唯一的 token，这个 token 在它们创建时由引用监视器生成。更确切地说，token 是一条大小为 16 字节的二进制数据，由 UUID 生成器随机产生，确保不会重复。而进程、文件则是由现有的操作系统标识符，也就是 pid 和 inode 标记。

值得指出的一个问题是，如果系统被关闭，系统状态如何存储。由于操作系统层次的 IFC 系统通常以进程为控制主体。一旦进程退出，或者系统被关闭，那么 IFC 系统就无法再确定标签集的归属。因为一个进程只有在它运行时才存在。如果进程退出再重新运行，就不能说此时的进程和此前的进程是同一个了。这将造成严重的问题——试想，假如一个文件被赋予了标签，而由于物理机重启，这个标签已经被所有新进程遗忘。那么这个文件将无法被任何进程访问。解决这个问题的通常思路是让引用监视器对进程做身份认证，可以用密码学的方法让不同时期的进程认证为同一个身份。但 DEfensor 使用了一种更简洁的方法，这种方法与 3.5

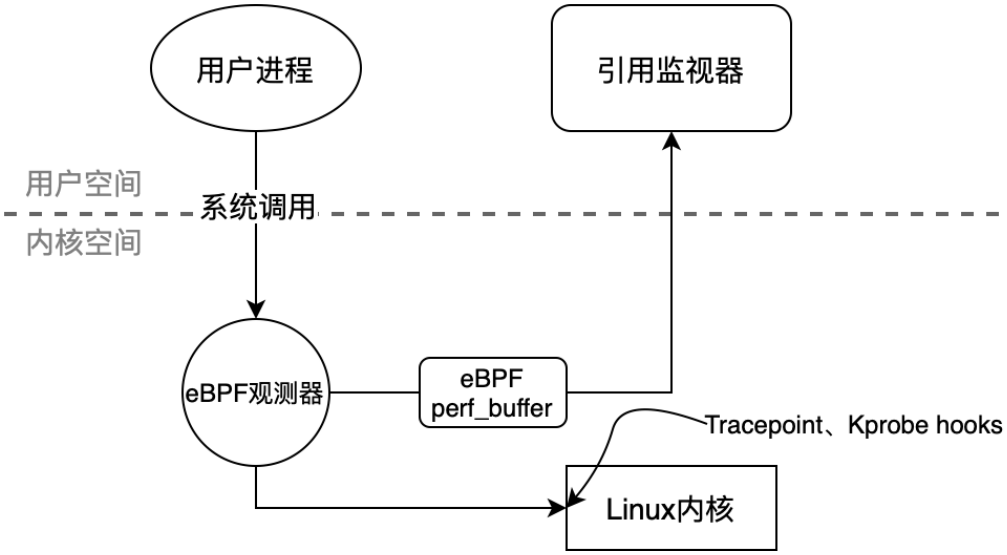

图 3.3　进程引用监视

节中描述的能力共享流程共用一个方案。意识到 DEfensor 系统存在的程序可以主动将自己能力集中的能力 token 存储到文件或其它位置，从而让能力持久化。下一次该程序运行时，可以获取文件中的 token 值，并用这个 token 值向引用监视器请求恢复能力。恢复能力后，程序也可以用能力获得对应的标签。这样，标签和能力可以被持久化存储。引用监视器只需保证标签数据库存储在硬盘上，而引用数据库则根据进程的创建与消失在内存中动态维护。

3.3　进程引用监视

为了控制进程的信息流，需要对进程建立的信道有所了解。这样我们就需要实时监视进程的引用情况。图 3.3 描述了引用监视的过程。

引用监视由进程行为触发，由 eBPF 观测器捕获并上报给引用监视器，引用监视器再根据这些行为改变响应状态。这其中最关键的就是 eBPF 观测器。eBPF 观测器通过 eBPF 集成的 Tracepoint、Kprobe 等钩子，抓取系统中所有的进程周期过程（包括进程的创建、复制、退出），以方便引用监视器管理标签集能力集的继承和销毁过程；同时抓取系统中信道的创建和修改过程（包括打开文件或创建管道等行为获得文件描述符，以及通过 dup、dup2 等函数修改文件描述符指向的信道等行为），以向引用监视器报告系统中实时的信道变化过程。

DEfensor 的进程引用监视流程由在用户空间运行的引用监视器和在内核运行的 eBPF 观测器协同运作。其为一种典型的系统钩子事件跟踪流程。当系统进程发生进程周期的变动（如 fork、vfork、clone、exit 等）时，由 eBPF 观测器设置的

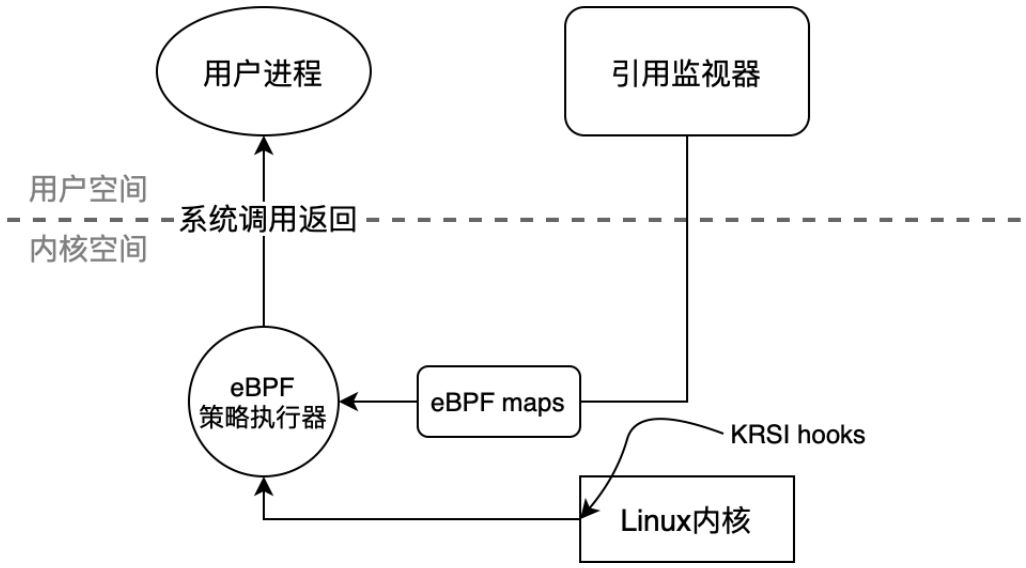

图 3.4　强制策略执行

tracepoint/sched 钩子和 tracepoint/syscall 钩子提取其函数参数，包括函数入口处关于进程 pid、uid、可执行文件路径、clone 选项等参数，以及函数出口可观测到的返回值（成功与否）和新产生的 pid 等参数；当系统进程发起信道的建立、删除和变动（如 open、close、pipe、dup、dup2 等）时，由 eBPF 观测器设置的 tracepoint/sched 钩子和 kprobe 钩子同样抓取其参数，包括函数入口处关于进程 pid、uid、可执行文件路径、打开文件路径、被操作的文件描述符等参数，以及函数出口处关于返回值、产生的文件描述符等参数。在抓取的过程中，运用 eBPF maps 对以上数据进行存储，之后，将这些系统钩子事件进行打包，由 eBPF 提供的 perf_buffer 将其发送到用户空间的引用监视器并以异步方式进行解包，之后进行分发并分析各事件代表的系统状态改变含义。

在用户进程产生进程周期变动和信道的建立、删除和变动时，eBPF 观测器都仅仅是对其进行跟踪和记录，不影响其执行过程。无论是受信任进程还是非受信任进程，eBPF 观测器对他们都是透明的。由于 eBPF 观测器运行在内核沙箱中，因此对于系统全局的进程行为的观测并不会过度地影响系统速度，保证了系统的执行效率。

3.4　强制策略执行

信息流控制系统不可缺少的一环就是强制策略执行。DEfensor 在获知进程引用的信道以及标签能力集后，就可以按照 2.3 节中的规则确定进程是否能对信道读或写。这个过程如图 3.4 所示。

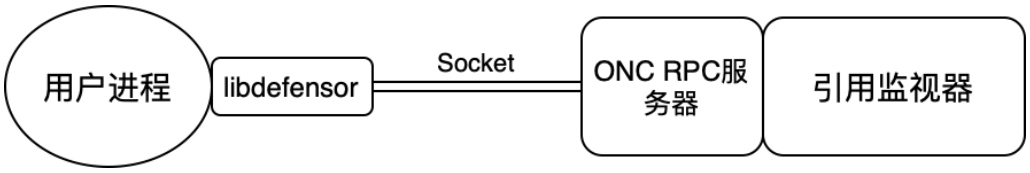

图 3.5　用户接口

每当进程的标签集、能力集改变，或进程创建、加入一个信道时，引用监视器就会刷新针对该进程的访问控制策略。这些访问控制策略以（信道，访问权限）的二元组表示。其中，信道由进程的 pid 和操作信道的文件描述符 fd 唯一确定。而访问权限有四种：不可读不可写、可读不可写、可写不可读、可读可写。执行权限不在考虑范围内，因为执行一段数据并不意味着信息的流动。为了加快策略匹配速度，默认情况下访问控制采用黑名单机制。即只记录那些禁止通信的信道策略，对没有记录的信道采用放行策略。这是因为当前大多数程序仍未为 DEfensor 而设计，它们之间的大多数信息流都是不受限制的，记录这些信道将会消耗大量的内核代价。

这些访问控制策略接下来会被写入到 eBPF maps 结构中，以传递给 eBPF 策略执行器。eBPF 策略执行器并不会对更新的访问控制策略做出响应，它只在需要时查询 maps 结构，获得正确的策略。策略执行器使用了最新的 KRSI 技术。KRSI 是一类专为系统运行时安全设计出的钩子，它事实上是一个 Linux 安全模块（LSM），允许 eBPF 程序加载到 LSM 钩子上，从而可以对 hook 到的行为做禁止或放行处理。eBPF 策略执行器挂载在 file_permission、inode_permission、ipc_permission 等 LSM 钩子上，从而在进程打开信道后、访问信道前执行。当进程触发这些 LSM 钩子时，eBPF 策略执行器会检查该进程以及该进程请求访问的信道，查询 maps 中的访问控制策略是否禁止这次读/写访问，并根据相应规则予以拦截或放行。

3.5　用户接口

DEfensor 的用户接口事实上是一个远程过程调用（RPC）服务器。如图 3.5 所示，引用监视器会启动一个线程运行 ONC RPC 服务器，这个服务器定义了一系列的 RPC 函数，接受客户端的连接与发送请求。同时，DEfensor 为用户提供了一个 API 库 libdefensor，它里面封装好了与 RPC 服务器建立连接和发送请求的过程。用户程序只需要引入该库，然后调用其中的 API 就可以向 DEfensor 引用监视器发送请求并接收返回结果。

用户能够发送的请求有创建标签、创建带标签的文件、根据能力修改标签、请求获得能力、放弃能力、查询标签集。事实上，用户的请求仅限于标签管理，其它的事务都由系统自动完成。这些请求会被用户接口移交给标签管理模块，标签管理模块进行一系列检查（如是否拥有增减标签的相应能力、修改的能力是否会影响信道访问策略等），然后将处理结果返回。

这其中一个重要的过程是进程之间传递能力。在许多情况下，进程需要将自己的能力传递给其它进程，这样其它进程才能获得相关权限。DEfensor 系统中有三种传递能力的方法：

- 进程创建标签时将标签的能力加入到全局能力集 O 中，这样所有的进程都会拥有这个标签的能力。
- 进程调用 fork 创建子进程，子进程会自动继承父进程所有的能力集和标签集。此时，子进程应当经过一系列标签变更和能力变更后调用 exec 运行其它程序。这样，其它程序也获得了部分继承的能力。这种方式适用于 DEfensor 对目标程序完全透明的情况。
- 如果进程与目标进程之间能够通信，进程将可以能力对应的 token 值经信道传输给对方。对方拿到 token，通过用户接口向引用监视器请求获得该能力。引用监视器经过验证就会允许它获得能力。这种方式适用于目标程序也针对 DEfensor 系统修改了源代码的情况。

需要注意的是，DEfensor 系统不允许直接的标签传递，也不允许进程不经对方主动申请就赋予对方能力。这样做既是为了防止控制系统的滥用，也是为了封堵通过标签传播信息的隐蔽信道。

3.6 示例

作为 DEfensor 的一个典型使用场景，接下来将描述 Bob 如何在 DEfensor 系统中使用 shell 来启动一个可能潜在恶意的编辑器。（如图 3.6）

Bob 的私密文件都被预先打上了 b 标签。首先，shell 作为一个受信的进程，Bob 可以确信 shell 将数据只导出到终端而不是其他地方。于是他启动 shell 并给予 shell $O_{sh} = \{b^+, b^-\}$ 的能力集，这意味着 shell 具备 b 标签的完全控制能力，可以随时在 S_{sh} 中增减 b 标签。shell 的标签集为空集，能与作为外部接收器的终端（也具有空标签集）通信。

然后，shell 通过 fork 和 exec 调用产生运行编辑器。fork 时，引用监视器观测

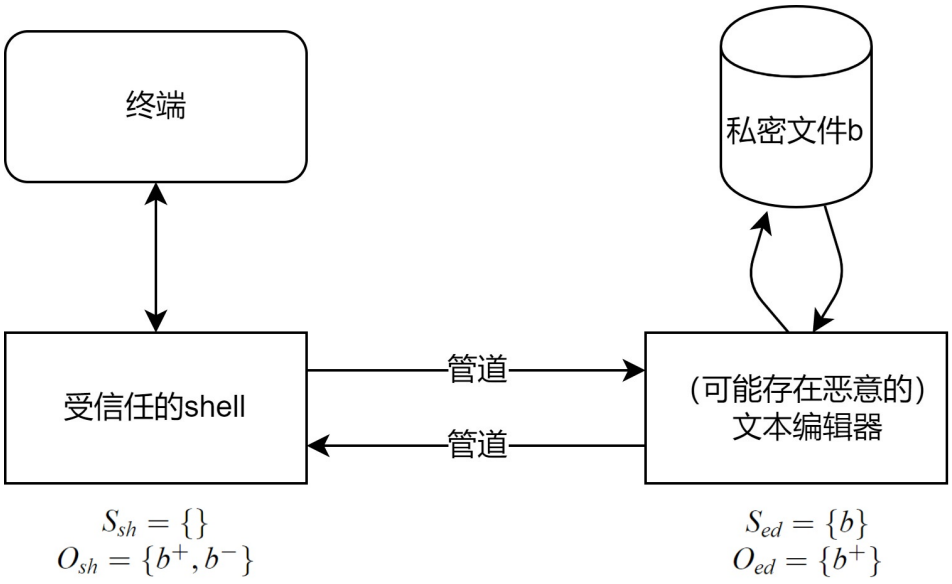

图 3.6　Bob 使用 shell 启动恶意编辑器

到新进程的产生，默认让其继承 shell 的标签集和能力集。shell 会在 fork 之后执行引用监视器提供的 drop_cap API，丢弃 b^- 能力。并执行 gain_tag API，利用 b^+ 能力获得 b 标签。这样，子进程的标签集为 $\{b\}$，能力集为 $\{b^-\}$。（假设全局能力集 O 为空）做完这些，子进程调用 exec 运行编辑器，编辑器的标签集和能力集由此被设置妥当。

与此同时，shell 还会建立与编辑器的标准输入输出流连接的管道，使编辑器的内容能够回显到 shell 上。管道由 shell 建立，因此属主是 shell 进程。所以管道的 $S_{pipe} =$ ，$O_{pipe} = \{b^+, b^-\}$. 而编辑器要加入管道，DEfensor 发现双方的私密标签集不一致，但管道有 b^+、b^- 能力，双方的通信是合法的。

最后，拥有 b 标签的编辑器有权打开 Bob 的私密文件，并将内容通过管道传输给 shell，由 shell 回显。如果潜在恶意的编辑器意图通过不受允许的信道向外传递私密文件，譬如建立一个连接因特网的 socket。引用监视器会发现信道的对方是外部接收者，标签集为空，于是设置禁止编辑器写该 socket 的规则。每当编辑器尝试往 socket 中写数据时，eBPF 策略执行器就会 hook 到这一行为，并予以禁止。这样就限制住了信息流的传出，保证了私密信息的安全。

4 总结

本文提出的 DEfensor 系统支持在现代操作系统中使用 DIFC 控制，通过 DEfensor，程序员可以为私密数据提供强大的安全性，并同时支持良好的性能和可移植性，即使应用程序中存在漏洞和恶意行为，也可实现信息流的控制，从而防御黑客通过恶意软件将信息通过不受信任的信道或者后门窃取信息。同时得益于 DIFC 的去中心化特性，在未来的工作中，DEfensor 系统还有希望考虑实现跨网络的 DIFC 系统，实现公司局域网层面（甚至广域网）的私密数据安全。我们将在未来的工作中进一步完善 DEfensor 系统，并与现存经典的 DIFC 系统进行性能对比测试和安全性评估。我们希望操作系统通过支持 DEfensor 系统并使其与旧的软件共存，使开发人员广泛接触和采纳 DIFC 风格的安全策略和程序设计技术，以提高现代操作系统的安全性。

参考文献

[1] MYERS A C, LISKOV B. A decentralized model for information flow control[J]. ACM SIGOPS Operating Systems Review, 1997.

[2] BELL L J, d. e. la padula. Secure computer system: Unified exposition and multics interpretation[J]. Mi Tre, 1976.

[3] BIBA K J. Integrity considerations for secure computer systems[J]. Electronic Systems Div Air Force Hanscom Afb, 1977.

[4] DENNING D E. A lattice model of secure information flow[J]. Communications of the ACM, 1976, 19(5): 236-243.

[5] MYERS A C, LISKOV B. Protecting privacy using the decentralized label model [J]. ACM, 2003.

[6] VANDEBOGART S, EFSTATHOPOULOS P, KOHLER E, et al. Labels and event processes in the asbestos operating system[J]. Acm Transactions on Computer Systems, 2007, 25(4): 11.

[7] ZELDOVICH N, BOYDWICKIZER S, KOHLER E, et al. Making information flow explicit in histar[J]. USENIX Association, 2006.

[8] FRASER T. Lomac: Low water-mark integrity protection for cots environments[J]. IEEE, 2000.

[9] MCILROY M, REEDS J A. Multilevel security in the unix tradition[J]. Software: Practice and Experience, 1992.

[10] GARFINKEL T, PFAFF B, ROSENBLUM M. Ostia: A delegating architecture for secure system call interposition[C]//Network and Distributed System Security Symposium. 2004.

[11] SEABORN M. Plash: tools for practical least privilege[M]. 2008.

[12] KROHN M, YIP A, BRODSKY M, et al. Information flow control for standard os abstractions[J]. ACM SIGOPS Operating Systems Review, 2007, 41(6): 321-334.